Journal of Universal Computer Science, vol. 16, no. 17 (2010), 2353-2373
submitted: 15/2/10, accepted: 30/8/10, appeared: 1/9/10 © J.UCS

Linking UML and MDD through UML Profiles: a Practical Approach based on the UML Association

Giovanni Giachetti, Manuela Albert, Beatriz Marín, Oscar Pastor
(Centro de Investigación en Métodos de Producción de Software
Universidad Politécnica de Valencia,
Camino de Vera s/n 46022 Valencia, Spain
{ggiachetti, malbert, bmarin, opastor}@pros.upv.es)

Abstract: In a model-driven development context, the definition (or selection) of an appropriate modeling language is a crucial task. OMG, in the model-driven architecture specification, recommends the use of UML for model-driven developments. However, the lack of semantic precision in UML has led to different model-driven approaches proposing their own domain-specific modeling languages in order to introduce their modeling needs. This paper focuses on customizing the UML association in order to facilitate its application in model-driven development environments. To do this, a well-defined process is defined to integrate the abstract syntax of a domain-specific modeling language that supports a precise semantics for the association construct in UML by means of the automatic generation of a UML profile. Finally, a brief example shows how the results obtained by the application of the proposed process can generate software products through a real model compilation tool.

Keywords: UML, Association, Profile, MDD, MDA, DSML
Categories: D.2.2, D.2.12, D.3.3, H.1.1, I.6.5

1 Introduction

The *Model Driven Development* (MDD) approach has achieved great relevance in the software industry, improving the software development process and reducing the cost of the developed applications [Völter, 07]. In this context, one of the most widely used approaches is the *Model Driven Architecture* (MDA) [Booch, 04] [OMG, 03], defined by OMG [OMG, 10b]. The MDA approach recommends the use of UML to define the conceptual models involved in MDD processes. However, UML is defined as a general purpose language with a flexible semantics that does not provide enough precision to define models that can be automatically transformed into complete software representations.

The association is one of the key constructs in UML for which a fully unambiguous semantics still does not exist [Milicev, 07]. In early versions of UML, many authors have reported this issue [Graham, 97] [Snoeck, 01]. In the most recent versions of UML (UML 2.0 and above), this semantics has been somewhat improved, but some precision problems still persist [Albert, 03] [Gueheneuc, 04]. For instance, the behavior related to creation, deletion, or update of association instances, or a complete semantics for the *aggregation* relationships are not clearly specified [France, 06].

In order to provide an effective solution for linking UML and MDD processes, this paper presents a proposal that allows the UML syntax (proposed in the UML specification) to be adapted to the modeling needs of specific MDD approaches. In particular, we advocate showing how to extend (customize) the abstract syntax of the UML constructs that are related to specifying association relationships among classes. This UML extension is carried out using a UML profile generated by applying a well-defined process, which is based on the definition of a particular metamodel that describes the abstract syntax required by the models of the reference MDD approach [Giachetti, 08]. To present our proposal, we have inherited the modeling aspects related to a specific MDD approach, the *OO-Method* approach [Pastor, 01]. We use this approach, since OO-Method is an object-oriented MDD method that has been successfully applied to the software industry[1].

This paper makes a twofold contribution: (1) it presents an industrially-tested semantics that can be used as a reference for the application of the UML association in MDD environments, and (2) it shows how a correct integration of the syntax that supports the proposed semantics can be performed by the application of a well-defined process [Giachetti, 09c], which is based on the standard UML extension mechanism, the UML profile. The paper also presents a brief example of how to obtain a final software product from a UML model that has been extended with the generated UML profile. This model compilation is performed using the industrial solution that implements the OO-Method approach [Gomez, 98].

The rest of the article is organized as follows: Section 2 presents a background of the concepts and technologies involved. Section 3 introduces the semantics adopted in this paper to improve the UML association. Section 4 shows how the customization of the UML association is performed. Section 5 presents a model compilation example related to a UML model that has been extended with the proposed semantics. Finally, Section 6 presents some conclusions and further work.

2 Background

This section is centered on the need to customize the UML specification for its appropriate application in MDD processes. Specifically, we show why the UML association must be adapted for this purpose. Additionally, a brief introduction about the OO-Method approach and UML profiles is also presented.

2.1 The UML Association

UML specifications include association definitions that do not achieve a consensus for a unified semantic definition. Several works [Diskin, 06] [Genova, 04] [Henderson-Sellers, 99b] [Milicev, 07] have appeared highlighting the drawbacks of the language and trying to answer many important questions concerning associations. With regard to the UML1.4 specification [OMG, 05], Henderson-Sellers has presented different works [Henderson-Sellers, 99a] [Henderson-Sellers, 99b] [Opdahl,

[1] OO-Method has been implemented in an industrial MDD suite called *Olivanova* [Pastor, 04], which has been developed by *CARE-Technologies* [CARE, 10] and applied in several companies such as Toshiba, Daimler-Chrysler, and Repsol.

01] searching for answers to some relevant questions, such as the directionality of associations or the special meaning of aggregation and composition. Special attention to the whole-part properties of the association has been given in [Barbier, 03] and [Belloir, 03]. Also, in [Stevens, 02], the author tries to clarify some confusing concepts regarding associations (without using formalizations), such as the use of *tuple* for defining links, some complex questions of the multiplicity definition, or the static and dynamic notion of associations. This last concept is also discussed in [Genova, 04], where the authors propose a new classification for associations. With regard to most recent versions of the UML specification [OMG, 09b], it is recognized that the association definition has been improved, but some problems still persist [Albert, 03] [Gueheneuc, 04]. For instance, [Diskin, 06] presents a framework to formally explain several confusing notions of associations and detects some flaws in the association part of the UML metamodel. In [Graham, 97], the author centers on newer concepts that are related to association ends in order to improve the expressiveness of the UML association. In addition, there exists a well-known gap between the conceptual representation of the UML association and its correct implementation in final software products [France, 06], which is a relevant issue for the correct application of UML in MDD processes. This situation is also present in some implementation proposals for the UML association such as [Akehurst, 06] [Gessenharter, 08], where elements represented at the implementation level have no correspondence at the conceptual level. In our proposal, we have centered our attention on a MDD approach called OO-Method.

The OO-Method approach puts into practice most of the ideas presented in the analyzed works. However, unlike most of the works that just focus on specific parts of the association definition, OO-Method provides a holistic view of the association. For instance, some works just face the composition definition, which is a subtype of the association; others focus on the notation for specifying associations; and others on the alternatives to implement associations. Instead, OO-Method integrates all these aspects to obtain a complete association specification. Moreover, even though some of the analyzed proposals have a certain level of technology support, they are mostly applied at the theoretical and academic levels. In contrast, the OO-Method approach has been successfully applied to the software industry, which demonstrates the effectiveness of this approach to support real software development projects. Thus, the rigorous semantics of the OO-Method association encourages the use of this approach to explain how this relevant UML construct can be customized for effective MDD application.

2.2 The OO-Method Approach

OO-Method is an MDD approach that separates the application and business logic from the platform technology, allowing automatic code generation from the conceptual representation of the software systems [Pastor, 01]. The OO-Method production process (Figure 1) is comprised of three models: the *Conceptual Model*, the *Execution Model*, and the *Implementation Model*.

The *OO-Method Conceptual Model* provides the expressiveness and precision required to correctly specify *Management Information Systems* (MIS). It captures the static and dynamic properties of the system in a *Class Model*, a *Dynamic Model*, and a *Functional Model*. The conceptual model also allows the specification of the user interfaces in an abstract way through the *Presentation Model*. These four models represent the different conceptual views of the intended system, which has all the details needed for the generation of the corresponding software application. The complete definition of the OO-Method Conceptual Model is presented in [Pastor, 07].

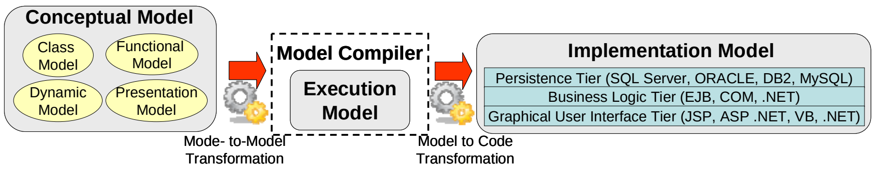

Conceptual Model
Class Model
Functional Model
Dynamic Model
Presentation Model
Model Compiler
Execution Model
Implementation Model
Persistence Tier (SQL Server, ORACLE, DB2, MySQL)
Business Logic Tier (EJB, COM, .NET)
Graphical User Interface Tier (JSP, ASP .NET, VB, .NET)
Mode- to-Model Transformation
Model to Code Transformation

Figure 1: OO-Method software production process

The class model is the core of the OO-Method conceptual model; the rest of the models involved are defined starting from elements of the class model. The constructs involved in the specification of associations among classes are defined within the OO-Method class model. Moreover, the correct specification of the different modeling aspects of the association is probably one of the most important and complex parts of the OO-Method class model. For this reason, the OO-Method association has been chosen to explain the linking approach presented in this article.

Thus, by integrating the modeling aspects of the OO-Method association into UML, we obtain an extended UML association that provides appropriate modeling information for its application in the OO-Method MDD process. To perform this integration according to the MDA guide [OMG, 03] and the UML specification [OMG, 09a] [OMG, 09b], we have considered the generation of a UML Profile.

2.3 The UML Profile Extension Mechanim

The UML profile extension mechanism is defined inside of the UML Infrastructure [OMG, 09a]. It defines the mechanisms used to adapt existing metamodels to specific platforms, domains, business objects, or software process modeling. Since this extension mechanism is a part of the UML standard, it can be supported by UML tools. This feature is one of the main advantages of the UML profile over other UML customization mechanisms [Bruck, 07], which are not part of the UML standard and, hence, are not supported by UML tools.

A UML profile is represented as a UML package that is stereotyped with the tag <<profile>>. It has three main constructs for the definition of the required extensions: stereotypes, tagged values, and OCL rules:

- The stereotype is the main construct for the specification of a UML profile. It is a special kind of UML class (specialization of the metaclass *Class* from the UML metamodel). Therefore, the semantics and notation of a stereotype are very similar to a UML class. The stereotypes are identified by a unique name and represent the set of the extensions that are applied over the classes of the extended metamodel. The extended classes are identified by means of extensions relationships that go from the stereotypes to the metaclasses that they extend.

- A tagged value is a property (specialization of the UML metaclass *Property*) that is owned by a stereotype. A tagged value represents a new property that is added to the metaclass extended by the corresponding stereotype. According to the last UML specifications (UML 2.1 and above), the type of the tagged values can be specified from data types, classes, and stereotypes. Therefore, the tagged values can be used for the definition of new attributes, and also for the definition of new associations.
- The OCL rules are defined by means of the Object Constraint Language [OMG, 10a]. Each OCL rule is related to a specific stereotype and is used to control the interaction among the different conceptual constructs (extended metaclasses). Even though the name OCL makes reference to the definition of constraints, according to the last OCL specification [OMG, 10a], OCL can also be used as a query language and as a language for the specification of functions and operations.

In general terms, the UML profiles that are present in the literature are manually elaborated without a well-defined process. This situation is motivated by the lack of a standard that specifies how the UML extensions must be defined [France, 06]. For this reason, many of the existent UML profiles are invalid or of poor quality [Selic, 07]. To avoid this situation, we apply a well-defined process that is based on the methodological solution introduced in [Giachetti, 09a], which consists in the automatic generation of a UML profile from a metamodel that describes the abstract syntax of the required conceptual constructs.

3 The Semantics Proposed to Customize the UML Association

This section introduces the semantics proposed to customize the UML association, which is inherited from the OO-Method approach. However, since many concepts in OO-Method already exist in the UML specification, we only focus on the aspects that meaningfully contribute to improving the UML association in the context of the MDD development process. [Marín, 08] shows a detailed case study of the OO-method approach, where the modeling flexibility that the proposed OO-Method semantics provides can be observed.

In the OO-Method context, an association is defined as a structural relationship between classes that represents connections (*links*) between the objects of these classes (*participant classes*). OO-Method associations are binary, so they only have one or two participant classes (one class in the recursive associations). Thus, the association concept used in this paper always refers to binary associations.

The *association ends* are the endpoints of an association, which connect the association to its participant classes. The name of an association end corresponds to the *role* of that end (the task that the participant class plays in the association). The association ends are characterized by the multiplicity property, which specifies the maximum/minimum number of objects that can/must be connected to an object of the opposite end. The relevant concepts that must be added to UML for appropriate definition of associations according to the OO-Method approach are:

- Unique identification for class instances (objects).

- Precise behavior for aggregation and composition concepts.
- Precise behavior related to creation, deletion and update of links.

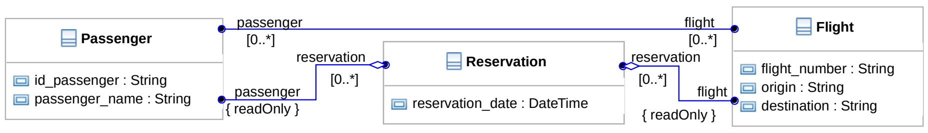


Figure 2: Example UML model

Figure 2 shows a brief UML model that is used throughout this paper to illustrate our proposal. This model was defined using the *Eclipse UML2* tool [Eclipse, 10b]. It shows an association between the classes *Passenger* and *Flight*, and an aggregation between these two classes and the class *Reservation*. A passenger can make a reservation for a specific flight, or can take a flight without a previous reservation. The association between the classes *Passenger* and *Flight* indicates those passengers that actually flew. Thus, a passenger with a reservation may not be related to a flight, for instance, if the passenger misses the flight.

The specific modeling features that are integrated into UML are presented below.

3.1 Object Identification

In UML, it is not possible to uniquely identify the objects participating in an association when the model is instantiated, since the UML specification [OMG, 09b] does not define a mechanism for the identification of class instances. It is interesting to observe that, even though the correct identification of objects is a relevant issue for correcting compilation of the association, proposals that deal with the compilation of the UML association usually omit this feature [Akehurst, 06] [Gessenharter, 08]. To solve this problem, the concept of *Identification Function* is introduced in the OO-Method approach.

The Identification Function corresponds to a set of structural properties that allows the unique identification of class instances. Thus, the Identification Function is specified by means of a set of attributes (one or more) owned by a class. In the example shown in Figure 2, the attribute *id_passenger* is a clear candidate to conform the Identification Function of the class *Passenger*; the same is true for the attribute *flight_number* of the class *Flight*. The Identification Function is specified by adding the Boolean property *isIdentifier* to the specification of class attributes. Thus, *isIdentifier* is set to *TRUE* when a class attribute participates in the Identification Function of the owning class.

3.2 Aggregation and Composition

In order to make the semantics of the aggregation concept more precise, we adopt the following UML assertion: "An association may represent an aggregation (i.e., a whole/part relationship). In this case, the association-end attached to the whole element is designated, and the other association-end of the association represents the parts of the aggregation. Only binary associations may be aggregations". In OO-Method, this definition is extended with additional semantics. Thus, the property of

Identification Dependency of the whole with regard to the part is introduced to represent the dependency that exists between composite (whole) and component (part) classes. This dependency is discussed in works such as [Barbier, 03].

The identification dependency implies that the identifier of the composite class is built using the identifier of the component classes (and, if necessary by adding some attribute of the composite class). According to UML, the lower cardinality of the association end related to the component class must be 1, that is, [1..x] cardinality. However, to guarantee the correct compilation of the identification dependency in the OO-Method development process, the upper cardinality of the association end related to the component classes is constrained to 1, that is, [1..1] cardinality. This constraint is defined since [1..*] cardinality implies that the identification function of the composite class must be conformed by a multi-valued attribute. However, the target implementation platforms of the OO-Method approach are based on relational databases that do not provide support for multi-valued attributes, such as SQL Server (see Figure 1). This situation is illustrated in Section 5.

In the aggregation example presented in Figure 2, the component classes are *Flight* and *Passenger* (with cardinality [1..1]) and the composite class is *Reservation* (with cardinality [0..*]). The Identification Function of *Reservation* is composed by the attribute *id_passenger* from *Passenger* and *flight_number* from *Flight*[2].

An aggregation can be specialized in a *composition (composite aggregation* in UML*)*, which presents additional features. These features are the following:

- A part must be included in, at most, one composite at a time.
- If a composite is deleted/modified, all its parts are deleted/modified with it.
- There is an identification dependency of the part with regard to the whole. Thus, for the composition, the cardinality related to the composite class must be [1..1]. Note that the identification dependency of composition is opposite to the identification dependency of the aggregation.

3.3 Creation, Deletion, and Modification of Links

The creation, deletion, and modification of links are only required in *dynamic* associations. A dynamic association is an association with two modifiable (*dynamic*) association ends. In a UML model, this can be represented as a binary association where the property *readOnly* of the two involved association ends is set as *false*. A dynamic association implies that its links can be changed (inserted, deleted, or updated) during the life of the participant objects.

In UML, it is necessary to manually define (in the participant classes) operations to represent the management (creation, deletion, and modification) of links. These operations must include the specification of the related behavior, which can be specified using different languages, such as natural language, *Action Semantics* [Sunye, 01] or the *Object Constraint Language* (OCL) [OMG, 10a]. For the definition of these operations, it must be taken into account that the management of links simultaneously affects properties of the two participant classes of the association. Therefore, the involved operations must be simultaneously executed in the participant classes. It is possible to observe that the definition of these operations is not trivial,

[2] The specification of the identification function for the class *Reservation* is not required, since it can be automatically inferred from the aggregated classes during the model compilation process.

and, hence, the manual definition of the behavior that is related to these operations makes the correct specification of the associations difficult and error-prone, which is of great relevance when this specification is interpreted by an automatic model compiler. To face this issue, certain works [Akehurst, 06] [Gessenharter, 08] have proposed a direct implementation of operations related to controlling the associations' behavior. In these proposals, the operations related to the management of links are represented at the implementation level (programming code). Nevertheless, since these operations do not have representation at the conceptual level, customization of the behavior related to links management cannot be performed in the UML model.

Thus, our proposal introduces the concept of *shared event* to represent these linking management operations. A shared event is a special kind of operation that defines the behavior related to dynamic associations. It is owned by the two participant classes, and its definition can be separately customized in each participant class. Thus, a shared event has a definition that is distributed between the classes that participate in the association. Events of this kind always require two input parameters, either linked objects or objects to be linked. A shared event can be of three types:

- Insert Event: creates a link between an object at one end of the association and an object at the opposite end.
- Delete Event: removes an existent link between an object at one end of the association and a related object at the opposite end.
- Edit Event: changes an existent link between an object at one end of the association and a related object at the opposite end.

The types of shared events depend on the cardinality of the association ends. Cardinality [1..1] in an association end prevents an existent link from being deleted or a new link from being created, that is, the execution of an insert or delete event is not possible. In this case, a link is established during the creation of an object at the opposite end, and it can only be deleted when one of the participant objects is deleted. Hence, the dynamic association can only be managed by an *edit shared event* because the only option is to change the existent link for another one. Thus, the edit shared event has the effect of a simultaneous execution of an insert and deletion event, which prevents the violation of the association cardinality. In any other case, the dynamic associations are managed by the *insert* and *delete* events.

The behavior of a shared event can be customized by defining preconditions and post-conditions, which must be specified by means of well-formed, first-order logic formulas (this is exemplified in section 5.3). The case study presented in [Marín, 08] provides more detailed examples about the customization and integration of shared events in more complex services.

4 Integration of the Proposed Semantics into UML

In this section, we integrate the semantics proposed (in the previous section) into UML by applying the process introduced in [Giachetti, 09a]. By means of this *Integration Process* we obtain: 1) a specific metamodel that defines the abstract syntax required for representing the proposed association semantics, and 2) the UML profile that integrates our proposal into UML. The metamodel that represents the required abstract syntax is defined as an *Integration Metamodel* according to the proposal presented in [Giachetti, 08]. This Integration Metamodel is an EMOF-based metamodel [OMG, 06a] that allows the defined abstract syntax to be integrated into UML by means of a UML profile that is automatically generated. This generation is performed according to the set of transformation rules presented in [Giachetti, 09a], which are oriented to obtain the corresponding UML Profile from the Integration Metamodel. The Integration Process is comprised of three steps (see Figure 3): 1) the definition of the Integration Metamodel; 2) the identification of the required UML extensions (throughout the comparison of the Integration Metamodel and the UML metamodel); and 3) the transformation of the Integration Metamodel into the corresponding UML profile. The last two steps can be automatically performed.

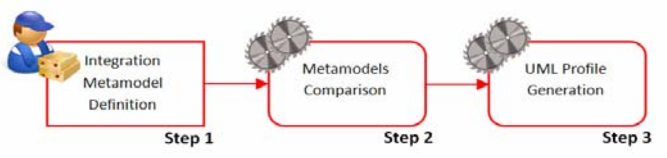


Figure 3: Process to integrate a DSML into UML

4.1 Defining the Integration Metamodel

The first step of the Integration Process consists of defining the Integration Metamodel to represent the abstract syntax of the modeling aspects that are required by the reference MDD approach. There are four conditions that an Integration Metamodel must hold for the automatic generation of the metamodel extensions. These are the following:

- All the classes from the Integration Metamodel are mapped to class of the UML Metamodel. This assures that the constructs from the MDD approach can be represented from the UML constructs.
- The mapping is defined between elements of the same type (classes with classes, attributes with attributes, and so on).
- An element from the Integration Metamodel is only mapped to one element of the UML Metamodel.
- If the properties of a class A from the Integration Metamodel are mapped to properties of a class B of the UML metamodel, then the class A is mapped to the class B or a specialization of it.

Figure 4 shows the Integration Metamodel that describes the abstract syntax for the semantics introduced in section 3. This corresponds to a subset of the whole metamodel used by the industrial implementation of the OO-Method MDD approach.

The Integration Metamodel presented has been specified using the Eclipse UML2 Tool [Eclipse, 10b] since it provides automatic generation of EMF metamodels from the defined UML2 metamodels. EMF [Budinsky, 03] is the *Eclipse Modeling Framework*, which is based on the EMOF specification. Also, the generated EMF metamodels are tagged with additional information to automatically obtain model editors that have interpreters for the defined OCL rules and that support UML profile extensions. Additionally, the Eclipse UML2 project provides a complete implementation of the UML metamodel, which is defined according to the official UML specification. This facilitates that the artifacts involved in the application of our proposal fulfill the OMG standards.

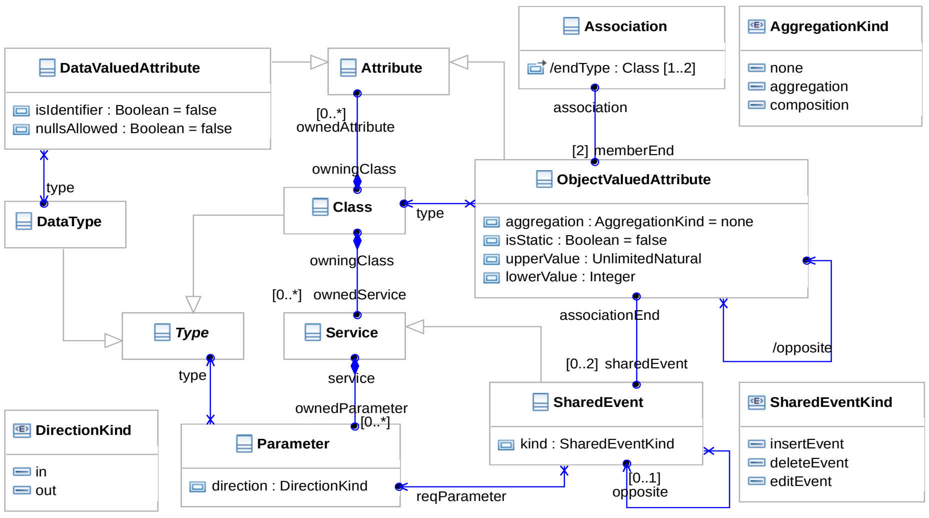


Figure 4: The Integration Metamodel of the proposed association semantics

According to the defined Integration Metamodel (Figure 4), the classes can own *data-valued* attributes (class *DataValuedAttribute*) or *object-valued* attributes (class *ObjectValuedAttributes*). The data-valued attributes are the typical class attributes, such as the name of the passenger in the UML example (Figure 2). In the class *DataValuedAttribute*, the attribute *isIdentifier* indicates whether the data-valued attribute participates as (part of) the identifier of its owning class (*isIdentifier* = True), and the attribute *nullsAllowed* specifies whether the data-valued attribute can take null values. A data-valued attribute that participates as an identifier of a class cannot take null values (*nullsAllowed* = False). An object-valued attribute represents an association end, such as the *passenger* and *flight* related to a *reservation* in the UML example of Figure 2, and it is related to the association by means of the association *memberEnd*. In the context of binary associations, the object-valued attributes must always have an opposite association end (association *opposite*). In the class *ObjectValuedAttributes*, the attribute *aggregation* indicates (if necessary) the kind of association (*aggregation*, *composition*, or *none*) related to the association end. The class related to an association end is indicated by the association *type*. In an object-valued attribute the attribute *isStatic* indicates if the association end is static (*isStatic* = True) or dynamic (*isStatic* = False). The name of the object-valued attribute (which is inherited from the class *NamedElement)* indicates the *role name* of the corresponding association end. The class *NamedElement* and the related inheritance hierarchy are not represented to simplify the Integration Metamodel diagram since all the classes defined are specializations of *NamedElement*.

Table 1 shows the OCL rules that object-valued attributes must fulfill to ensure that the cardinality constraint related to the Identification Dependency feature of aggregation and composition is not violated.

Description	If the attribute *aggregation* of an association end holds the value *#aggregation*, then the opposite end must have cardinality [1..1].
Context	ObjectValuedAttribute
OCL	self.aggregation = #aggregation implies self.opposite.lowerValue = 1 and self.opposite.upperValue = 1
Description	If the attribute *aggregation* of an association end holds the value *#composition*, this end must have cardinality [1..1].
Context	ObjectValuedAttribute
OCL	self.aggregation = #composition implies self.lowerValue = 1 and self.upperValue = 1

Table 1: OCL constraints for association ends

Description	Shared events can only be defined when both association ends are dynamic.
Context	ObjectValuedAttribute
OCL	self.temporality = #static or self.opposite.temporality = #static implies self.sharedEvent->isEmpty()
Description	Only the shared event *editEvent* can be defined for dynamic associations when one of the association ends has cardinality [1..1].
Context	ObjectValuedAttribute
OCL	((self.temporality = #dynamic) and (self.opposite.temporality = #dynamic) and (self.lowerValue = 1) and (self.upperValue = 1)) implies self.sharedEvent.kind = #editEvent and self.sharedEvent->size() = 1
Description	The *insert* and *delete* shared events must be defined for dynamic associations when both association ends have cardinality [x..*].
Context	ObjectValuedAttribute
OCL	((self.temporality = #dynamic) and (self.opposite.temporality = #dynamic) and (self.upperValue > 1) and (self.opposite.upperValue > 1)) implies self.sharedEvent->size() = 2 and self.sharedEvent->exists(se | se.kind = #insertEvent) and self.sharedEvent->exists(se | se.kind = #deleteEvent)
Description	A shared event requires an opposite shared event with the same name and kind, except in the case of recursive associations.
Context	SharedEvent
OCL	self.associationEnd.type <> self.associationEnd.opposite.type implies self.opposite->notEmpty() and self.kind = sel.opposite.kind and self.name = self.opposite.name
Description	In recursive associations, a shared event does not have an opposite event.
Context	SharedEvent
OCL	(self.associationEnd.type = self.associationEnd.opposite.type) implies self.opposite->isEmpty()
Description	A shared event cannot be opposite to itself.
Context	SharedEvent
OCL	self.opposite->notEmpty() implies self <> self.opposite

Table 2: OCL constraints for shared events

The definition of a shared event is distributed between the classes that participate in the association because a shared event simultaneously changes properties of the participant objects. To support this semantics, in the Integration Metamodel a shared event is represented as two dependent events, which are related by means of the association *opposite*. Table 2 shows the OCL constraints defined for shared events.

A shared event must always participate in an association and is only related to one object-valued attribute. This is represented by the association *associationEnd* with the cardinality [1..1].

The association *ownedParameter* (inherited from *Service*) represents the parameters of a shared event. One of the two parameters that are required by a shared event (the two participant objects) is the object that executes the service, whose type corresponds to the class that owns the service. However, this parameter does not need to be defined in the model because it is implicit in the semantics of the service and it can be inferred from the association *owningClass*. The second parameter required by a shared event (the participant object of the opposite association end) is identified by the association *reqParameter*.

According to the Integration Process, to complete the definition of the Integration Metamodel it is necessary to identify the equivalences between this metamodel and the UML metamodel. Table 3 presents these equivalences.

Integration Metamodel	**UML Metamodel**	**Integration Metamodel**	**UML Metamodel**
AggregationKind	**AggregationKind**	**ObjectValuedAttribute**	**Property**
.none	.none	.type	.type
.aggregation	.shared	.association	.association
.composition	.composite	.opposite	.opposite
Association	**Association**	.aggregation	.aggregation
.memberEnd	.memberEnd	.isStatic	.isReadOnly
Attribute	**Property**	.upperValue	.upper
.owningClass	.class	.lowerValue	.lower
Class	**Class**	**Parameter**	**Parameter**
.ownedAttribute	.ownedAttribute	.direction	.direction
.ownedService	.ownedOperation	.service	.operation
DataType	**DataType**	.type	.type
DataValuedAttribute	**Property**	**Service**	**Operation**
.type	.type	.owningClass	.class
DirectionKind	**ParameterDirectionKind**	.ownedParameter	.ownedParameter
. in	.in	**SharedEvent**	**Operation**
.out	.out	**Type**	**Type**

Table 3: Equivalences between the Integration Metamodel and the UML Metamodel

4.2 Comparing the Integration Metamodel and the UML Metamodel

The second step of the Integration Process requires a comparison between the Integration Metamodel and the UML Metamodel to be performed. It allows the identification of the differences between the Integration Metamodel and the UML Metamodel. These differences correspond to the metamodel extensions that must be implemented in the UML profile generation. With regard to the conditions established for the Integration Metamodel definition and the mapping information presented in Table 3, this second step of the process is automatically performed. This prevents the extra time, effort, and potential errors that are involved in a manual identification of the required extensions. Table 4 summarizes the results of this comparison for the presented example.

Integration Metamodel	Difference
Association	
.memberEnd	lower bound (IM = 2; UML = *)
DataValuedAttribute	
.type	type (IM = DataType; UML = Type)
.isIdentifier	*new*
.nullsAllowed	*new*
ObjectValuedAttribute	
.type	type (IM = Class; UML = Type)
.association	lower bound (IM = 1; UML = 0)
.opposite	lower bound (IM = 1; UML = 0)
.sharedEvent	*new*
SharedEvent	
.kind	*new*
.opposite	*new*
.reqParameter	*new*
.associationEnd	*new*
SharedEventKind	*new*

Table 4: Comparison between the Integration Metamodel and the UML Metamodel

In Table 4, the column *Integration Metamodel* shows the *elements* of the Integration Metamodel that differ from the UML elements, and the column *Difference* shows what the differences are by indicating the values that differ for the Integration Metamodel element (*IM*) and the UML element (*UML*). The word *new* in the column *Difference* indicates when the Integration Metamodel introduces an element that does not exist in UML. Thus, the elements that must be introduced into UML to solve the identified differences are the extensions that must be defined in the UML profile.

4.3 Generating the UML Profile

The third and last step of the Integration Process corresponds to the generation of the final UML profile (Figure 5). The UML profile generation is performed by means of a set of transformation rules (explained in [Giachetti, 09a]) that are applied over the Integration Metamodel. These rules are applied taking into account the equivalences presented in Table 3 and the differences presented in Table 4. The main features of these transformation rules are the following:

- Equivalent classes of the Integration Metamodel are transformed into stereotypes that extend the corresponding UML class identified in Table 3. If the class of the Integration Metamodel has the same name as the corresponding UML class, then a prefix is added to differentiate the name of the stereotype from the name of the extended class. In the example, the prefix *OOm* is used. For instance, the class *Association* of the Integration Metamodel generates the stereotype *OOmAssociation* (prefix + class name).
- The *new* properties (attributes and associations) that are identified in the metamodel comparison (see Table 4) are represented as tagged values. For instance, the attribute *isIdentifier* of the class *DataValuedAttribute* is defined as a tagged value in the stereotype *DataValuedAttribute* (see Figure 5).
- Differences between equivalent properties are managed with OCL constraints. For instance, the lower bound difference of the associations *opposite* and *association* of the class *ObjectValuedAttribute* (see Table 4) are managed with an OCL rule with the following structure:

  ```
  self.[property]->size() >= [newLowerBound]
  ```

  The OCL rules are defined in the stereotypes that are generated from the involved classes; in this case, the stereotype *ObjectValuedAttribute*.

- The generated stereotypes have all the constraints defined in the transformed classes. For each constraint, the elements of the Integration Metamodel are replaced by the corresponding elements of the UML metamodel or by elements of the generated UML profile in the case of new elements. For instance, the first OCL rule defined in Table 1 is defined in the stereotype *ObjectValuedAttribute* as follows:

  ```
  self.aggregation = #shared implies
  self.opposite.lower = 1 and self.opposite.upper = 1
  ```

  In the presented OCL rule, the elements written in *italics* indicate the UML elements that are used to replace the corresponding Integration Metamodel elements according to the equivalences presented in Table 3.

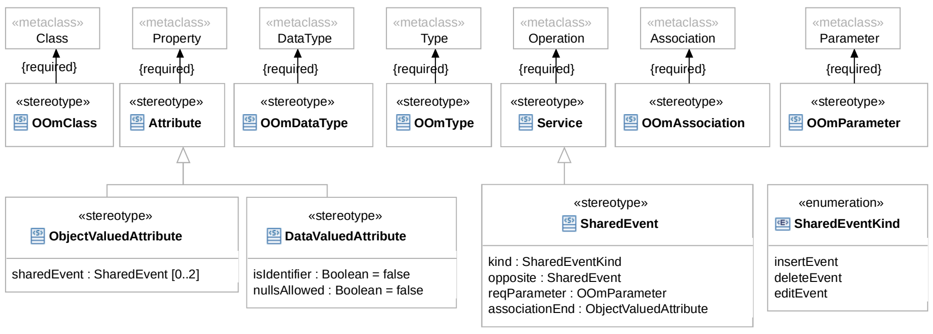
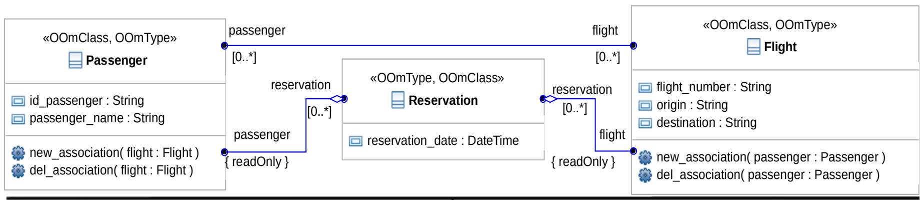
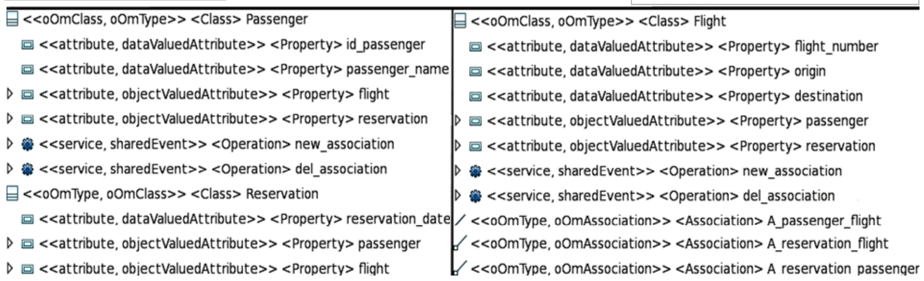


Figure 5: UML Profile generated from the defined Integration Metamodel

5 Compiling the Extended UML Association

Figure 6 shows the example UML model extended with the generated UML profile and the tabular representation of this model, where the application of the different stereotypes can be observed. The services for the management of links between objects are defined as *shared events* by means of the stereotype *sharedEvent*, the service *new_association* is defined as an *insert event*; and the service *del_association* is defined as a *delete event*.




<<oOmClass, oOmType>> <Class> Passenger
<<attribute, dataValuedAttribute>> <Property> id_passenger
<<attribute, dataValuedAttribute>> <Property> passenger_name
<<attribute, objectValuedAttribute>> <Property> flight
<<attribute, objectValuedAttribute>> <Property> reservation
<<service, sharedEvent>> <Operation> new_association
<<service, sharedEvent>> <Operation> del_association
<<oOmType, oOmClass>> <Class> Reservation
<<attribute, dataValuedAttribute>> <Property> reservation_date
<<attribute, objectValuedAttribute>> <Property> passenger
<<attribute, objectValuedAttribute>> <Property> flight

<<oOmClass, oOmType>> <Class> Flight
<<attribute, dataValuedAttribute>> <Property> flight_number
<<attribute, dataValuedAttribute>> <Property> origin
<<attribute, dataValuedAttribute>> <Property> destination
<<attribute, objectValuedAttribute>> <Property> passenger
<<attribute, objectValuedAttribute>> <Property> reservation
<<service, sharedEvent>> <Operation> new_association
<<service, sharedEvent>> <Operation> del_association
<<oOmType, oOmAssociation>> <Association> A_passenger_flight
<<oOmType, oOmAssociation>> <Association> A_reservation_flight
<<oOmType, oOmAssociation>> <Association> A_reservation_passenger

Figure 6: Example UML model extended with the generated UML profile

Once the UML model is correctly specified, it is compiled with the OO-Method model compilation technology [Pastor, 04] by using a specific interchange proposal [Giachetti, 09b], which is based on the generated UML profile and the mapping information obtained during the application of the integration process. In the compilation of the UML model, default services for the creation, deletion, and edition of instances are automatically created for each class. These default services are not created for those classes that already have these kinds of services.

It is important to remark that the extensions introduced in the UML model provide a precise definition for the association at the conceptual level, which allows the independence between the business logic and the implementation platforms. With regard to this independence, the OO-Method compilation technology can generate applications for different implementation platforms from the same UML model. For instance, from an OO-Method model, the industrial OO-Method implementation can currently generate software products in *.Net*, and *J2EE* developing platforms, and *Oracle*, *SQL Server*, *DB2*, and *PostgreSQL* database servers (see Figure 1). The case study presented in [Marín, 08] shows how the target platform is selected by using a specific configuration tool for the model compilation process. We have selected the .Net platform (.Net 2.0 framework and C# language) and SQL Server database as implementation technologies for the example in this paper.

5.1 Compilation of the object identification

The Identification Function has a direct impact on the database generated in the compilation of the UML model (see Figure 7). In this database, the identification function of each class is transformed into a primary key of the table that corresponds to the compiled class. In addition, the table *TM_PassengerFlight* (not present in the UML model) has been automatically generated for the implementation of the many-to-many association.

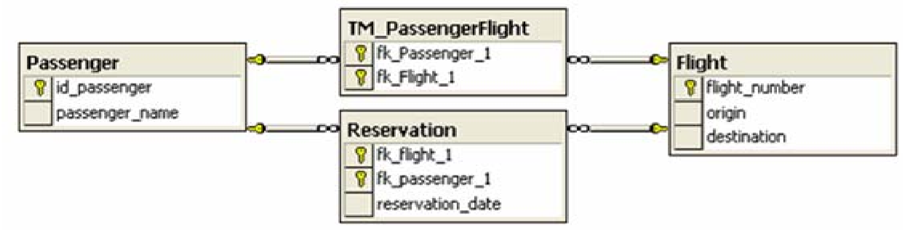
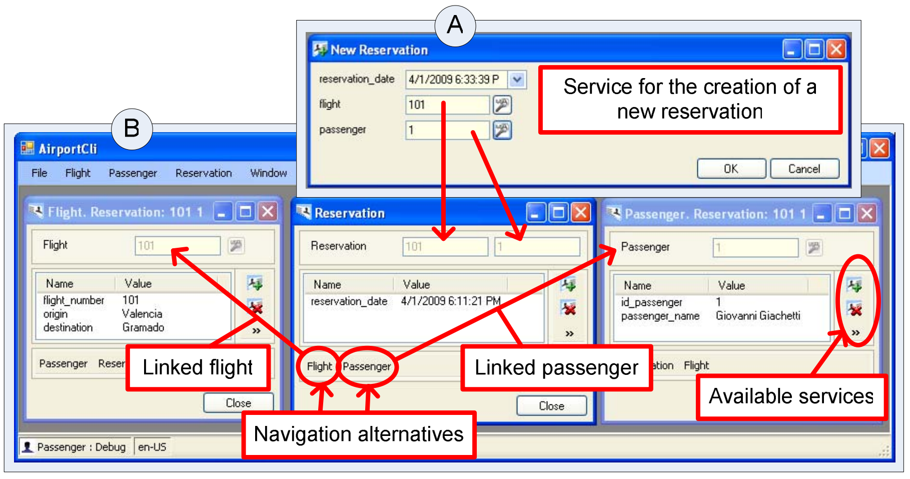


Figure 7: Relational model of the database generated from the UML model example

5.2 Compilation of the Aggregation

In the example, the class *Reservation* is aggregated by *Flight* and *Passenger*. The aggregation implies the identification dependency of the whole with regard to the part. Figure 7 shows that to implement the identification dependency, the primary key of the table *Reservation* is comprised of the primary keys of the tables *Passenger* and *Flight*. This implementation is automatically inferred by the model compilation process from the defined aggregation relationships. In addition, the cardinality [1..1] in the component side is mandatory according to the semantics proposed for the identification dependency.


New Reservation
reservation_date 4/1/2009 6:33:39 P
flight 101
passenger 1

Flight. Reservation: 101 1
Flight 101

Name	Value
flight_number	101
origin	Valencia
destination	Gramado

Reservation
Reservation 101 1

Name	Value
reservation_date	4/1/2009 6:11:21 PM

Passenger. Reservation: 101 1
Passenger 1

Name	Value
id_passenger	1
passenger_name	Giovanni Giachetti

Figure 8: Execution of the service that creates new instances of the class Reservation

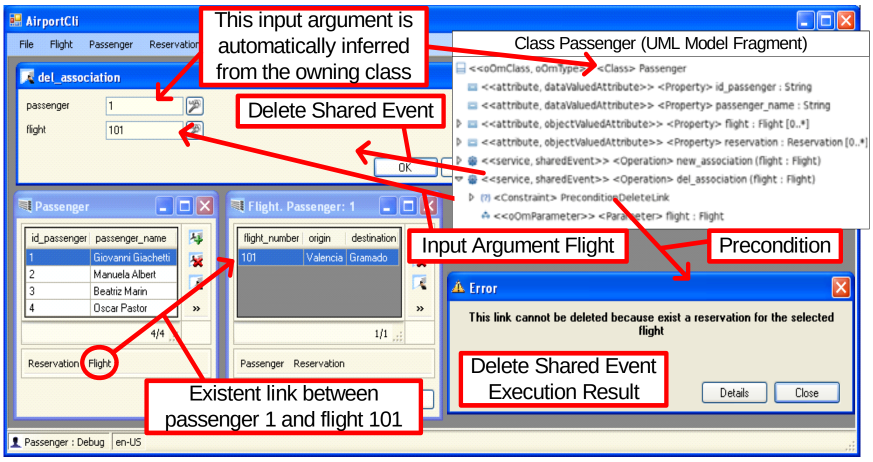
The cardinality [1..1], which is defined for the association ends related to the classes *Flight* and *Passenger*, implies that a reservation must always be related to a flight and a passenger. Thus, the links between a reservation and the corresponding passenger and flight must be created at the same time as the creation of the reservation. This particular semantics of the aggregation must be considered in the compilation of the service that creates instances of the class *Reservation*. Figure 8 shows a screenshot of the application generated from the UML model, which is related to the execution of the service for creation of new reservations.

Figure 8-A shows the form related to the execution of the service, and Figure 8-B shows the result of the execution, which indicates the flight and the passenger linked during the creation of the reservation. The navigation alternatives have been inferred from the associations defined in the UML model.

Figure 8-A also shows that the service for the creation of an instance of the class *Reservation* has three inbound arguments: the date of the reservation (defined in the UML model) and the flight and passenger related to the new reservation (inferred from the aggregations relationships).

5.3 Compilation of shared events

Figure 9 shows a screenshot of the application related to the execution of the shared event *del_association* (executed from an instance of *Passenger).* This shared event has two input arguments: the identifiers of the linked passenger and flight. The execution of the shared event *del_association* destroys a link that exists between the selected objects (the input arguments).


AirportCli
File Flight Passenger Reservation
This input argument is automatically inferred from the owning class
Delete Shared Event
del_association
OK
Class Passenger (UML Model Fragment)
Passenger
Flight. Passenger: 1
Input Argument Flight
Precondition
Error
Delete Shared Event Execution Result
Details
Close
Reservation Flight
Passenger Reservation
Existent link between passenger 1 and flight 101
Passenger : Debug
en-US

passenger	1
flight	101

id_passenger	passenger_name
1	Giovanni Giachetti
2	Manuela Albert
3	Beatriz Marin
4	Oscar Pastor

4/4

flight_number	origin	destination
101	Valencia	Gramado

1/1

```
<<oOmClass, oOmType>> <Class> Passenger
  <<attribute, dataValuedAttribute>> <Property> id_passenger : String
  <<attribute, dataValuedAttribute>> <Property> passenger_name : String
  <<attribute, objectValuedAttribute>> <Property> flight : Flight [0..*]
  <<attribute, objectValuedAttribute>> <Property> reservation : Reservation [0..*]
  <<service, sharedEvent>> <Operation> new_association (flight : Flight)
  <<service, sharedEvent>> <Operation> del_association (flight : Flight)
    (?) <Constraint> PreconditionDeleteLink
    <<oOmParameter>> <Parameter> flight : Flight
```

This link cannot be deleted because exist a reservation for the selected flight

Figure 9: Execution of the service del_association with a precondition

Specific behavior for the management of links is defined at the conceptual level by means of the customization of the defined shared events. Thus, in the shared event *del_association*, we define the precondition: *a link between a passenger and a flight cannot be destroyed if there already exists a reservation related to these two objects.*

Figure 9 shows that the execution of the service *del_association* (from an instance of the class *Passenger*) does not fulfill the precondition (already exists a reservation for the selected passenger and flight); therefore, an error-message is displayed. This situation shows the relevance of the Identification Function, since the identifiers of the classes *Passenger* and *Flight* are used in the detection of pre-existent links.

6 Conclusions and Further Work

This article has presented two main contributions. The first of these is the application of a well-defined process for linking UML and MDD approaches in order to allow the automatic compilation of UML models by using existent MDD technologies. In particular, we advocate customizing the semantics of the UML association with a precise semantics that is obtained from an industrially applied MDD approach [Pastor, 07]. The abstract syntax that supports this semantics is defined using an Integration Metamodel [Giachetti, 08], which allows the automatic generation of a UML profile that integrates the conceptual constructs and properties required for the proposed semantics into UML. This reduces the complexity related to the correct specification of UML extensions, providing an advantage over a manual UML profile specification, which is a time-consuming and error-prone task [Selic, 07]. Furthermore, the proposed process takes advantage of the OMG standards and existent open-source tools such as [Jouault, 08][Eclipse, 10a][Eclipse, 10b], which facilitate the interchange of knowledge within the MDD community.

The second main contribution of this paper is the semantics proposed for the UML association and the UML extensions generated to support this semantics. These extensions provide the capability of precisely representing the structure and behavior of the association at the conceptual level, which can be used to automatically obtain a software product in different implementation platforms. This distinguishes our proposal from others that provide a direct implementation of the UML association in a specific language [Akehurst, 06] [Gessenharter, 08], which require a specific behavior for the associations to be defined directly in the code. Thus, the proposed association modeling representation can be used as a reference by different MDD approaches.

As future work, we plan the construction of a complete modeling suite for OO-Method, which will be implemented on the Eclipse UML2 open-source tool following the proposed linking approach. Thus, it will be possible to obtain an effective MDD solution that takes advantage of the UML specification and the current open-source MDD technologies that are based on the OMG standards.

Acknowledgments

This work has been developed with the support of MEC under the project SESAMO TIN2007-62894 and GVA ORCA PROMETEO/2009/015.

References

[Akehurst, 06] Akehurst, D.H., Howells, W.G.J., McDonald-Maier, K.D.: Implementing Associations: UML 2.0 to Java 5. Journal of Software and Systems Modeling, vol. 6 nº 1, 1–33 (2006)

[Albert, 03] Albert, M., Pelechano, V., Fons, J., Ruiz, M., Pastor, O.: Implementing UML Association, Aggregation, and Composition. A Particular Interpretation Based on a Multidimensional Framework In: 15th Conference on Advanced Information Systems Engineering (CAISE'03), pp. 143–158 (2003)

[Barbier, 03] Barbier, F., Henderson-Sellers, B., Le Parc-Lacayrelle, A., Bruel, J.-M.: Formalization of the Whole-Part Relationship in the Unified Modeling Language. IEEE Transactions on Software Engineering, vol. 29 nº 5, 459–470 (2003)

[Belloir, 03] Belloir, N., Bruel, J.-M., Barbier, F.: Whole-Part Relationships for Software Component Combination. In: 29th EUROMICRO Conference (EUROMICRO'03), pp. 86–91. IEEE Computer Society (2003)

[Booch, 04] Booch, G., Brown, A.W., Iyengar, S., Rumbaugh, J., Selic, B.: An MDA Manifesto. Business Process Trends/MDA Journal (2004)

[Bruck, 07] Bruck, J., Hussey, K.: Customizing UML: Which Technique is Right for You? IBM (2007)

[Budinsky, 03] Budinsky, F., Brodsky, S.A., Merks, E.: Eclipse Modeling Framework, isbn: 0131425420. Pearson Education (2003)

[CARE, 10] CARE-Technologies Web Page, http://www.care-t.com (2010)

[Diskin, 06] Diskin, Z., Dingel, J.: Mappings, Maps and Tables: Towards Formal Semantics for Associations in UML2. In: MoDELS 2006, pp. 230–244. LNCS (2006)

[Eclipse, 10a] Eclipse Foundation: Eclipse Model Development Tools Project, http://www.eclipse.org/modeling/mdt/ (2010)

[Eclipse, 10b] Eclipse Foundation: Eclipse UML2 Project, http://www.eclipse.org/uml2/ (2010)

[France, 06] France, R.B., Ghosh, S., Dinh-Trong, T., Solberg, A.: Model-driven development using UML 2.0: Promises and pitfalls. In: IEEE Computer, vol. 39 nº 2, 59–66 (2006)

[Genova, 04] Génova, G., Llorens, J., Fuentes, J.M.: UML Associations: A Structural and Contextual View Journal of Object Technology, vol. 3 nº 7 (2004)

[Gessenharter, 08] Gessenharter, D.: Mapping the UML2 Semantics of Associations to Java Code Generation Model In: MODELS 2008. LNCS, pp. 813–827. (2008)

[Giachetti, 08] Giachetti, G., Valverde, F., Pastor, O.: Improving Automatic UML2 Profile Generation for MDA Industrial Development. 4th International Workshop on Foundations and Practices of UML (FP-UML) – ER Workshop, vol. LNCS 5232, pp. 113–122. Springer (2008)

[Giachetti, 09a] Giachetti, G., Marín, B., Pastor, O.: Using UML as a Domain-Specific Modeling Language: A Proposal for Automatic Generation of UML Profiles. In: CAiSE'09, vol. LNCS 5565, pp. 110–124. Springer (2009)

[Giachetti, 09b] Giachetti, G., Marín, B., Pastor, O.: Using UML Profiles to Interchange DSML and UML Models. In: Third International Conference on Research Challenges in Information Science (RCIS), pp. 385–394. IEEE Computer Society (2009)

[Giachetti, 09c] Giachetti, G., Marin, B., Pastor, O.: Integration of Domain-Specific Modeling Languages and UML through UML Profile Extension Mechanism. International Journal of Computer Science and Applications, vol. 6 nº 5, 145–174 (2009)

[Gomez, 98] Gómez, J., Insfrán, E., Pelechano, V., Pastor, O.: The Execution Model: a component-based architecture to generate software components from conceptual models. In: Workshop on Component-based Information Systems Engineering (1998)

[Graham, 97] Graham, I., Bischof, J., Henderson-Sellers, B.: Associations considered a bad thing. Journal of Object-oriented Programming vol. 9 nº 9, 1–48 (1997)

[Gueheneuc, 04] Guéhéneuc, Y., Albin-Amiot, H.: Recovering binary class relationships: Putting icing on the UML cake In: Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'04), pp. 301–314 (2004)

[Henderson-Sellers, 99a] Henderson-Sellers, B., & Barbier, F.: What is this thing called aggregation? In: TOOLS 29, pp. pp. 216–230. IEEE Computer Society (1999)

[Henderson-Sellers, 99b] Henderson-Sellers, B., Barbier, F.: Black and white diamonds. In: UML'99, pp. 550–565. LNCS (1999)

[Jouault, 08] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming, vol. 72 nº 1-2, 31–39 (2008)

[Marín, 08] Marín, B., Giachetti, G., Pastor, O.: The Photography Agency: A case study of the OO-Method Approach. Technical Report DSIC-II/13/08, Universidad Politécnica de Valencia, Valencia, España (2008)

[Milicev, 07] Milicév, D.: On the Semantics of Associations and Association Ends in UML. IIEE Transactions on Software Engineering, vol. 33 nº 4 (2007)

[OMG, 10a] OMG: Object Constraint Language 2.2 Specification (2010)

[OMG, 10b] OMG: Object Management Group Web site, http://www.omg.org/ (2010)

[OMG, 09a] OMG: UML 2.2 Infrastructure Specification (2009)

[OMG, 09b] OMG: UML 2.2 Superstructure Specification (2009)

[OMG, 06a] OMG: MOF 2.0 Core Specification (2006)

[OMG, 05] OMG: UML 1.4.2 Specification (2005)

[OMG, 03] OMG: MDA Guide Version 1.0.1 (2003)

[Opdahl, 01] Opdahl, A.L., Henderson-Sellers, B., Barbier, F.: Ontological analysis of whole-part relationships in OO-models. Information and Software Technology nº 43, 387–399 (2001)

[Pastor, 01] Pastor, O., Gómez, J., Insfrán, E., Pelechano, V.: The OO-Method Approach for Information Systems Modelling: From Object-Oriented Conceptual Modeling to Automated Programming. In: Information Systems. Elsevier Science, vol. 26 nº 7, 507–534 (2001)

[Pastor, 04] Pastor, O., Molina, J.C., Iborra, E.: Automated production of fully functional applications with OlivaNova Model Execution. ERCIM News nº 57 (2004)

[Pastor, 07] Pastor, O., Molina, J.C.: Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling. 1st edition, isbn: 978-3-540-71867-3. Springer, New York (2007)

[Selic, 07] Selic, B.: A Systematic Approach to Domain-Specific Language Design Using UML. In: 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), pp. 2–9 (2007)

[Snoeck, 01] Snoeck, M., Dedene, G.: Core modeling concepts to define aggregation. L'objet vol. 7 nº 3 281–306 (2001)

[Stevens, 02] Stevens, P.: On the interpretation of binary associations in the Unified Modelling Language. Journal on Software and System Modeling, vol. 1, 68–79 (2002)

[Sunye, 01] Sunyé, G., Pennaneac'h, F., Ho, W.-M., Guennec, A.L., Jézéquel, J.-M.: Using UML Action Semantics for Executable Modeling and Beyond. In: CAiSE 2001, vol. LNCS 2068, pp. 433–447. Springer (2001)

[Völter, 07] Völter, M., Stahl, T., Bettin, J., Haase, A., Helsen, S., Czarnecki, K.: Model-Driven Software Development: Technology, Engineering, Management. Wiley (2007)